# SIR-Trading

## Security review

Version 1.0

Reviewed by
**nmirchev8**
**deth**

# Table of Contents

**1 About Egis Security** **3**

**2 Disclaimer** **3**

**3 Risk classification** **3**
3.1 Impact . . . 3
3.2 Likelihood . . . 3
3.3 Actions required by severity level . . . 3

**4 Executive summary** **4**

**5 Findings** **5**
5.1 High risk . . . 5
5.1.1 `_payAuctionWinner` decodes does not handle correctly failed transfer call 5
5.1.2 APE clones domain separator is violated and the exploiter can use it in his advantage . . . 6
5.1.3 Exploiter can steal depositors WETH from vault using them for sir distribution . 7
5.2 Medium risk . . . 9
5.2.1 Stepwise jump farming in `Staking` contract . . . 9
5.2.2 If an auction token is paused, winner will loose his opportunity to claim it . . . 10
5.3 Low risk . . . 11
5.3.1 Incorrect price for negative ticks due to lack of rounding down . . . 11
5.3.2 If the probed fee tier is better, we update `oracleState` with that tier, but `tickPriceX42` stays to the value from the last tier . . . 11
5.4 Informational . . . 12
5.4.1 Pool cardinality is only increased once every 25 hours . . . 12

# 1 About Egis Security

Egis Security is a team of experienced smart contract researchers, who strive to provide the best smart contract security services possible to DeFi protocols.

The team has a proven track record on public auditing platforms like Code4rena, Sherlock, and Cantina, earning top placements and rewards exceeding $170,000. They have identified over 150 high and medium-severity vulnerabilities in both public contests and private audits.

# 2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

# 3 Risk classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likelihood: High | Critical | High | Medium |
| Likelihood: Medium | High | Medium | Low |
| Likelihood: Low | Medium | Low | Low |

## 3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

## 3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

## 3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

# 4 Executive summary

**Overview**

| | |
|---|---|
| Project Name | SIR-Trading |
| Repository | Private |
| Commit hash | 4c43aa188381806f08f77f5af7681e2a9318d93c |
| Resolution | e0ac239654a70473dce8007ce72e1c1302922681 |
| Documentation | https://docs.sir.trading/ |
| Methods | Manual review |

**Scope**

| |
|---|
| src/APE.sol |
| src/Oracle.sol |
| src/SIR.sol |
| src/Staker.sol |
| src/SystemControl.sol |
| src/SystemControlAccess.sol |
| src/SystemState.sol |
| src/TEA.sol |
| src/Vault.sol |
| src/libraries/* |

**Issues Found**

| | |
|---|---|
| Critical risk | 0 |
| High risk | 3 |
| Medium risk | 2 |
| Low risk | 2 |
| Informational | 1 |

# 5 Findings

## 5.1 High risk

### 5.1.1 `_payAuctionWinner` decodes does not handle correctly failed transfer call

**Severity:** *High risk*

**Context:** Staker.sol#L524-L525

**Description:**

`_payAuctionWinner` makes a low-level `call` to transfer token funds:

```
        /** Pay the winner if tokenAmount > 0
            Low-level call to avoid revert in case the destination has been banned
                from receiving tokens.
         */
        (success, data) = token.call(abi.encodeWithSignature("transfer(address,
            uint256)", auction.bidder, tokenAmount));

        /** By the ERC20 standard, the transfer may go through without reverting (
            success == true),
            but if it returns a boolean that is false, the transfer actually failed.
         */
        if (data.length > 0 && !abi.decode(data, (bool))) return false;
```

As we can see the comment states that the desired behavior is to catch a potential revert, without reverting the main function call. However, this is not the case and the transaction will revert when trying to decode returned `data` to `bool` in case of revert. When a token transfer reverts with an error message, we will have `data.length > 4` with the error message and `success = false`. The following may lead to the DoS of the following auctions for the given token if the recipient is blocklisted.

**NOTE** That there are some tokens that revert on zero amount transfer, which means that if the bidder has claimed his reward using `payAuctionWinner` function, later on, `collectFeesAndStartAuction` will try to transfer 0 amount, which will result in an unhandled revert.

Another impact is that the auction cannot be started with `collectFeesAndStartAuction` for the first time because the transaction will try to transfer 0 funds to `address(0)`, which reverts in almost every `erc20` token. Users won't have an incentive to start bidding because fees cannot be claimed until the end of the first auction.

**Recommendation:** Introduce:

```
if (success == false) return false;
```

**Resolution:** Fixed

### 5.1.2 APE clones domain separator is violated and the exploiter can use it in his advantage

**Severity:** *High risk*

**Context:** APE.sol#L53-L57

**Description:** Apes tokens will be deployed using `ClonesWithImmutableArgs` library to safe gas. This means that we have an implementation contract - `APE.sol`, which defines immutable args in it's constructor:

```
    constructor() {
        INITIAL_CHAIN_ID = block.chainid;
        INITIAL_DOMAIN_SEPARATOR = _computeDomainSeparator();
    }
```

Then we will deploy clone proxies, which will `delegatecall` to this implementation contract. Immuntable arguments defined in the implementation will be the same for all proxies because they are defined in the bytecode of the contract (implementation contract). We can notice that we define `INITIAL_DOMAIN_SEPARATOR` in the constructor, which will use the implementation address as `verifyingContract` and empty string as `name`:

```
    function _computeDomainSeparator() private view returns (bytes32) {
        return
            keccak256(
                abi.encode(
                    keccak256("EIP712Domain(string name,string version,uint256
                        chainId,address verifyingContract)"),
                    keccak256(bytes(name)),
                    keccak256("1"),
                    block.chainid,
                    address(this)
                )
            );
    }
```

When we deploy a new proxy we have a root problem:

- A permit message hash should be used with the original `DOMAIN_SEPARATOR` (of the implementation contract).
- If a user signs such a message for a `spender`, the spender can double spend the signature, if the owner has this balance for another ape proxy **Recommendation:**

Make `INITIAL_DOMAIN_SEPARATOR` state var and initialize it in `initialize` function.

**Resolution:** Fixed by calculating custom ape `INITIAL_DOMAIN_SEPARATOR` as immutable argument.

### 5.1.3 Exploiter can steal depositors WETH from vault using them for sir distribution

**Severity:** *High risk*

**Context:** Vault.sol#L308-L309

**Description:**

In `Vault.sol` there is a functionality to swap debt token for the collateral token. We will directly call `pool.swap` function and then execute the `uniswapV3SwapCallback` to mint ape/tea and adjust reserves/state. An exploiter can manipulate this functionality by calling `Staker::collectFeesAndStartAuction` inside of his `onERC1155Received` implementation if he uses a vault that has `WETH` as a debt token. The issue is that `withdrawFees` will use the weth amount that user provide for the swap as a fee amount because we don't increase the reserves for the debt token:

```
function withdrawFees(address token) external returns (uint256
    totalFeesToStakers) {
    require(msg.sender == _SIR);

    // Surplus above totalReserves is fees to stakers
    totalFeesToStakers = IERC20(token).balanceOf(address(this)) - totalReserves[
        token];

    if (totalFeesToStakers != 0) {
        TransferHelper.safeTransfer(token, _SIR, totalFeesToStakers);
    }
}
```

Then `uniswapV3SwapCallback` will transfer this weth amount to the uniswap pool to finish the swap. This will result in leaving `weth.balanceOf(vault)< totalReserves[token]`, effectively stealing honest depositors funds.

Imagine the following scenario: We have two vaults in `Vault.sol` debt : collateral 1st -> weth : wbtc 2nd -> usdc : weth

with following reserves:

- weth = 10e18
- wbtc = 1e18
- An exploiter contract that implements `onERC1155Received` calls `mint` providing 1st pool params, and paying 5e18 eth in native asset.
- This will wrap 1e18 eth into weth => having vault weth balance = 15e18
- The flow continues, we enter uniswap pool swap => we enter `uniswapV3SwapCallback` func and calculate gentleman corresponding X amount of WBTC that he is depositing to the pool
- We then calculate and mint his corresponding TEA tokens and call `minter.onERC1155Received`.
- Above function is implemented from the exploiter to call `Staker::collectFeesAndStartAuction` for weth, which will calculate 15e18 (weth balance of vault before transfering tokens for the swap to the pool) - 10e18 = 5e18 as `totalFeesToStakers` and transfer this amount to stakers
- Then we continue by transferring 5e18 weth to the uniswap pool for the swap, leaving vault weth balance to 5e18 and reserves still equal to 10e18

**Resolution:** Fixed by transfering weth prior to `_mint`

## 5.2 Medium risk

### 5.2.1 Stepwise jump farming in `Staking` contract

**Severity:** *Medium risk*

**Context:** Staker.sol#L425-L426

**Description:**

`Staking` contract can withdraw accumulated fees from `vault` contract at most once every 10 days. We call `_distributeDividends` inside of `collectFeesAndStartAuction` , or `payAuctionWinner`. `_distributeDividends` increases `stakingParams.cumulativeETHPerSIRx80`, meaning that stakers before that operation can now claim rewards. The problem is that there is no staker lock period, so exploiters can front-run any `_distributeDividends` transaction call, stake their sir, claim the reward and unstake the tokens. They can repeat the process for each reward distribution. The following will result in inflated rewards for honest stakers, while the exploiter provides just-in-time liquidity.

**NOTE** that there is a possibility of combining the attack vector with a flashloan, if `SIR` token has gained such liquidity on uniswap for example. This way exploiter can combine in one transaction `staking -> distributing rewards -> claiming -> unstaking` and repaying the flash loan. The following exploit introduces a major reward share manipulation problem, which leads to reward theft from the honest stakers.

**Recommendation:** Consider introducing a mandatory lock period for the stakers.

**Resolution:** Fixed

### 5.2.2 If an auction token is paused, winner will loose his opportunity to claim it

**Severity:** *Medium risk*

**Context:** Staker.sol#L426-L427

**Description:** Anyone can call `collectFeesAndStartAuction`, which will try to transfer the current contract balance to the auction winner, but if the transfer fails, the transaction will continue by resetting the auction. The following is not fair for the auction winner because he lost his WETH to bid, but didn't receive a reward. The token transfer may revert due to the token being paused or if the bidder is blocklisted.

**Recommendation:**

- Consider making `payAuctionWinner` only callable by the auction winner, so he can specify a recipient address (this will mitigate the problem of having a blocklisted bidder)
- Another solution is to implement a refund mechanism which should refund bidder's weth, if the token transfer reverts.

**Resolution:** Fixed

## 5.3 Low risk

### 5.3.1 Incorrect price for negative ticks due to lack of rounding down

**Severity:** *Low risk*

**Context:** Oracle.sol#L534

**Description:** If `(tickCumulatives[1] - tickCumulatives[0])` is negative, the tick should be rounded down as it's done in the OracleLibrary from uniswap:

```
int56 tickCumulativesDelta = tickCumulatives[1] - tickCumulatives[0];
uint160 secondsPerLiquidityCumulativesDelta =
    secondsPerLiquidityCumulativeX128s[1] -
        secondsPerLiquidityCumulativeX128s[0];

arithmeticMeanTick = int24(tickCumulativesDelta / secondsAgo);
// Always round to negative infinity
if (tickCumulativesDelta < 0 && (tickCumulativesDelta % secondsAgo != 0))
    arithmeticMeanTick--;
```

In case `(tickCumulatives[1] - tickCumulatives[0])` is negative and `(tickCumulatives[1] - tickCumulatives[0])% secondsAgo != 0`, then returned tick will be bigger then it should be, hence incorrect prices would be used.

**Recommendation:** Implement the following line: `if (tickCumulativesDelta < 0 && ( tickCumulativesDelta % secondsAgo != 0))arithmeticMeanTick--;`

**Resolution:** Acknowledged

### 5.3.2 If the probed fee tier is better, we update `oracleState` with that tier, but `tickPriceX42` stays to the value from the last tier

**Severity:** *Low risk*

**Context:** Oracle.sol#L356-L361

**Description:** In `Oracle::updateOracleState` we will first fetch oracle TWAP price from the current `OracleState` params and then check if the duration for fee tier update has passed and update it, if so. If the new fee tier passes all requirements/checks, we will update the oracle state `uniswapFeeTier` and `indexFeeTier`, but `tickPriceX42` will be the value obtained from the previous pool.

**Recommendation:** Consider first checking/updating the fee tier and then updating the price in `updateOracleState` flow

**Resolution:** Acknowledged

## 5.4 Informational

### 5.4.1 Pool cardinality is only increased once every 25 hours

**Severity:** *Informational*

**Context:** Oracle.sol#L374-L378

**Description:** According to the protocol documentation, oracle autonomous mechanism should be increasing the cardinality (when it is below the TWAP_DURATION) on each mint/burn: "To address this, the system incrementally extends the TWAP duration each time a mint/burn transaction occurs.", However, in the current implementation `updateOracleState` function will internally calculate `oracleState.cardinalityToIncrease`, but only increase it if we compare the fee tiers (once every 25 hours):

```
            if (block.timestamp >= oracleState.timeStampFeeTier +
                DURATION_UPDATE_FEE_TIER) {
                // No OF because timeStampFeeTier is uint40 and constant
                    DURATION_UPDATE_FEE_TIER is a small number
                bool checkCardinalityCurrentFeeTier;
```

Considering that we may be increasing the cardinality for different tiers each time (if the probed score is larger), the time required to achieve the full `TWAP_DURATION` coverage will significantly increase.

**Resolution:** Acknowledged